INFER: INTERACTIVE TIMING PROFILES BASED ON BAYESIAN NETWORKS [1]

Michael Zolda[2]

Abstract
We propose an approach for timing analysis of software-based embedded computer systems that builds on the established probabilistic framework of Bayesian networks. We envision an approach where we take (1) an abstract description of the control flow within a piece of software, and (2) a set of run-time traces, which are combined into a Bayesian network that can be seen as an interactive timing profile. The obtained profile can be used by the embedded systems engineer not only to obtain a probabilistic estimate of the WCET, but also to run interactive timing simulations, or to automatically identify software configurations that are likely to evoke noteworthy timing behavior, like, e.g., high variances of execution times, and which are therefore candidates for further inspection.

Keywords: Bayesian networks, embedded systems, hardware modeling, measurement-based execution time analysis, software modeling, probabilistic modeling, profiling, real-time systems

1. Introduction

With the increasing number of embedded computer systems in everyday life applications, like cars, digital entertainment systems, or mobile phones, knowledge about the real physical behavior of such systems is becoming more and more important. In particular, when a computer system is embedded in a real physical process, like in an engine control system, or a digital media stream decoder, knowledge of its timing properties becomes crucial.

Despite the notable advances in timing analysis of embedded computer systems, performing a WCET analysis of state-of-the-art systems is becoming more difficult, as intricate processor features, like caches, pipelining, and branch prediction, trickle down from the desktop and server to the embedded processor domain. As a result, such systems are becoming too complex for a complete, detailed analysis.

Whenever we have to reason about systems the complexity of which prohibits us from explicitly dealing with each special case, but the details are too important to simply ignore them, it is common and acknowledged practice to use models that *summarize* the impact of exceptions.

Probabilistic network models [10] provide a theoretically sound framework for summarizing complex relationships as probabilistic dependencies. Besides providing mechanisms for simulating the model's behavior under various user-defined scenarios, probabilistic network models offer a better traceability

[2]Institut für Technische Informatik, Technische Universität Wien, Treitlstraße 3/182/1, A-1040 Wien, Austria, e-mail: michaelz@vmars.tuwien.ac.at

[1]The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13.

of effects, compared to traditional regression models.

We propose an approach that uses Bayesian networks to model program execution times of software-based embedded systems. The structure of our networks is based on the structural properties of the soft- and/or hardware under test. The network is subsequently parameterized with empirical data obtained from run-time measurements on the actual target hardware.

2. State Machine Models

2.1. Reactive Systems

Depending on their method of operation, embedded computer systems can be classified into *reactive systems* and *transformative systems*. Whereas a reactive system keeps on running continuously, interacting with its physical environment, a transformative system is characterized by a "one-shot" mode of operation: the system starts by taking all its inputs, performs some calculation, and terminates with some output. Transformative systems are popular basic building blocks for more complex systems, where they are invoked periodically by a scheduler. If the transformative system is part of a (hard or soft) real time system [5], then it must fulfill certain previously specified timing constraints. Not meeting those timing constraints is considered a system failure.

We consider software-based transformative systems. Even though there have been various proposals to make such systems time-predictable by design, software that was written by applying traditional programming techniques, and/or which is running on modern processors has a highly unpredictable temporal behavior [4].

When we take a mathematical point of view on such a system, we essentially face an extremely complex state machine, where each state corresponds to an intricate hardware configuration, and where the transitions correspond to the change from one such configuration to another. Since each change between states corresponds to a real physical processes, we can associate an execution time with each transition. At this level of detail, the system is deterministic, and the execution time for each transition is cycle-accurate.

Unfortunately, we usually cannot describe a real transformative system in all details, for various reasons, like, e.g., inadequate detail of the available hardware specification, or sheer size. So we have to abstract our state machine model (SMM).

We consider three forms of abstraction, which are performed in the given order: *State elimination*, *Existential abstraction* and *segment abstraction*.

2.2. State Elimination

State elimination is achieved by removing states from the SMM. When a state s is removed, each of its incoming edges is redirected to the unique successor node s'[1]. Subsequently, the transition between s and s' is dropped, after adding its associated execution time to each of the redirected edges.

[1]The uniqueness is guaranteed by our premise of determinism of the SMM. From this uniqueness, it also follows that the SMM is free of cycles, because we are modeling a transformative system that is supposed to terminate with some output.

2.3. Existential Model Abstraction

Existential abstraction is achieved by collecting the concrete states of the SMM into sets, and viewing these sets as the abstract states of an *abstract state machine model (ASMM)*. The transitions of the ASMM are induced by may-semantics: There is an abstract transition between two abstract states of the ASMM, iff there is at least one concrete transition between two concrete states of the underlying SMM. Accordingly, the execution times of all the concrete transitions are collected into a multi-set with a corresponding may-semantics.

For software written in imperative programming languages, we usually consider *basic blocks* and their static control flow structure. In that case, transitions are identified with basic blocks, and abstract states are identified with basic block numbers.

Figure 1(a) gives an example source code written in the C programming language. Figure 1(b) shows the structure of the corresponding ASMM.

2.4. Segment Model Abstraction

Segment Abstraction is another form of abstraction, normally performed after existential abstraction. Whereas in existential abstraction we collect states into abstract states, in segment abstraction we collect paths into *segments*.

A segment is characterized by its entry/exit interface. It essentially binds together all paths that start at the entry interface (which is given as a collection of states) and end at the exit interface (another collection of states).

The utility of segments is threefold:

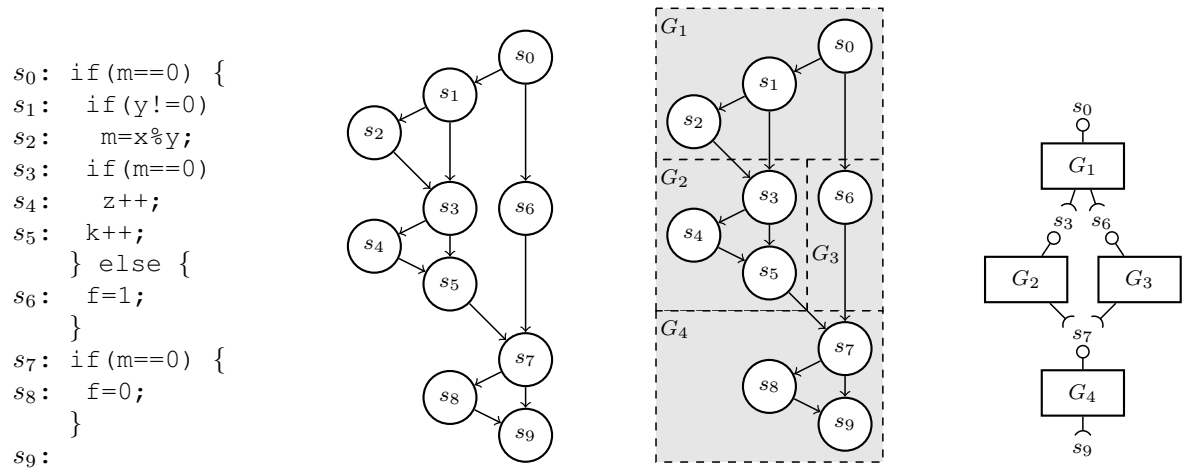
Handling of path explosion. The execution time of an individual basic block generally depends on the execution history. As a consequence, the execution times of individual basic blocks are not composable without loss of information. If the executions times are, for example, given as probability distributions, combining them through a convolution operator might lead to over- and underestimation of the probabilities of certain execution times. Even more importantly, in the measurement-based approach individual measuring of basic blocks may totally miss execution times that depend on a rare system state that is only reached through certain execution histories.

For maximal accuracy w.r.t. the ASMM, in the measurement-based approach, we would have to measure the execution time of each ASMM path. However, the number of ASMM paths is typically prohibitively large.

Segments are used to specify restricted measurement/coverage regions. They should be chosen such as to contain a limited number of paths, thus alleviating the path explosion problem. However, the paths within the segments should not be too short, such as not to sacrifice too much history-dependent information. A good segmentation algorithm should balance these two opposed goals.

Dependencies across segments can be captured/modeled by the Bayesian network model.

Obtaining accurate measurements. When we want to obtain the execution time information through measurement, we face the problem that some measurement methods are intrusive in the sense that the



(a) Example source code (b) Abstract state machine model (ASMM) (c) ASMM with segmentation (d) Abstract view on the segmentation

Figure 1. Illustration of an abstract state machine model (ASMM) and its segmentation

act of measuring affects the result. A typical example of this is measuring using software instrumentations. Such instrumentations affect the hardware state and may therefore also the affect the execution time of subsequent code. Secondly, some measurement methods can only provide limited accuracy, e.g. if they employ a timer with low granularity.

Both effects can be alleviated by avoiding the measuring of short sequences of instructions. Segments are used to specify measurement spans of a minimal length that is specific to the applied measurement method.

Logical Structuring. We do not only want to model the overall execution time behavior of the entire program, but also want to get more insight into the timing interactions within the program.

Segment are used to structure the program into smaller parts whose execution time behavior is made visible to the user.

For example, the engineer might consider a small loop as basic unit of functionality. Then this loop would be a candidate for a segment.

A *segment* is an ordered pair $\langle E, X \rangle$, where E is a set of *entry states*, and X is a set of *exit states*. Semantically, E is related to X through the segment $\langle E, X \rangle$, iff, for each state in E, there is a directed path to some state in X that does not touch any other state in X.

The set of *associated paths through* a segment $\langle E, X \rangle$ is the set of all paths[2] $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} s_n$, such that $s_0 \in E$, $s_n \in X$, and $s_i \notin X$, for any $0 < i < n$ and any $n > 0$.

Figure 1(c) indicates the partitioning of the ASMM from Figure 1(b) into four segments. For example, segment G_1 is the segment with $E = \{s_0\}$ and $X = \{s_3, s_6\}$. We can reach s_3 from s_0, e.g. via s_1, without touching s_6, and we can reach s_6 directly from s_0. The paths associated through G_1 are $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$, $s_0 \rightarrow s_1 \rightarrow s_3$, and $s_0 \rightarrow s_6$. Figure 1(d) provides a more abstract view of the

[2]We use labels to distinguish parallel edges. If there are no parallel edges, we can omit them.

segments and their interconnection. Note how the entry and exit states act as "connectors" between segments.

The idea behind segments is to abstractly and implicitly specify the set of paths that can be taken through a given region simply as an entry/exit interface. We then associate, with each segment, the multi-set of execution times of all its associated paths.

A segment is an over-approximation of the underlying subgraph of the SMM, in the sense that it summarizes all possible paths through that subgraph, and thus all possible path execution times. Segments should thus be seen as the primitives of a program with which we can associate meaningful timing information.

Segment abstraction can be performed by iteratively replacing subgraphs of the ASMM with matching segments, until all transitions have been collected into segments.

For a given ASMM, the choice of segments is not unique. We do not, at this moment, provide a concrete algorithm that incorporates all the requirements on segments that were discussed in this section.

3. Probabilistic Modeling

3.1. Bayesian Network Essentials

In probability theory, we consider *random experiments*, i.e., experiments with an outcome that is governed by some indeterministic mechanism. The possible outcomes of such an experiment are called *events*. To express a certain degree of confidence that a certain event will occur, real numbers from the interval $[0 \ldots 1]$ are used. Greater values indicate greater confidence; a value of 1 indicates absolute confidence that an event will occur, 0 indicates absolute confidence that an event will fail to occur, 0.5 indicates total indifference about the occurrence of an event. These real numbers are called *probabilities*.

A *random variable* is a variable whose value depends on the outcome of a random experiment. Through this dependency, random variables inherit probability values from the probabilities of their underlying events. Consequently, constraints over random variables can themselves be viewed as events.

When performing probabilistic reasoning, we always consider a specific *probabilistic model* that describes a set of random variables, as well as their qualitative and quantitative interconnections.

The *conditional probability* $P(X{=}x \mid Y_1{=}y_1, \ldots Y_n{=}y_n)$ designates the probability that variable X obtain the value x, given the *a-priori knowledge* that Y_i obtain the value y_i, for $1 \leq i \leq n$. Note that the order of the conditions is irrelevant, but that generally $P(X{=}x \mid Y{=}y, \ldots) \neq P(Y{=}y \mid X{=}x, \ldots)$. The *unconditional probability* $P(X{=}x)$ designates the probability that variable X obtain the value x, given no further information about the outcomes of any other variables in the model.

Two variables X and Y are *conditionally independent given a set of variables* $Y_1, \ldots, Y_n$, iff $P(X{=}x \mid Y{=}y, Y_1{=}y_1, \ldots Y_n{=}y_n) = P(X{=}x \mid Y_1{=}y_1, \ldots Y_n{=}y_n)$. Intuitively speaking, the outcome of Y does not influence the outcome of X, under the given a priori knowledge. Conditional independence

	Success	¬Success
Happiness	0.6	0.3
¬ Happiness	0.4	0.7

	Effort		¬Effort	
	Ideas	¬Ideas	Ideas	¬Ideas
Success	0.7	0.4	0.3	0.2
¬ Success	0.3	0.6	0.7	0.8

Effort	0.5
¬ Effort	0.5

Ideas	0.5
¬ Ideas	0.5

Figure 2. Example of a bayesian network

is a symmetric relation.

Bayesian network (BN) are a representation mechanism that can express the important class of *causal probabilistic models*. In a BN, the model's variables are expressed as nodes of a graph. For each variable X, the BN has a conditional probability table (CPT) that specifies the conditional probabilities $P(X{=}x \mid Y_1{=}y_1, \ldots Y_n{=}y_n)$ over all possible combinations of outcomes of its parent variables $Y_1, \ldots Y_n$.

The connection between a BN and the represented probabilistic model is completed by the following formal property: Any variable X is conditionally independent of all its non-descendants, given its parents $\mathbf{Y} = \{Y_1, \ldots Y_n\}$, and no subset of $\mathbf{Y}$ satisfies this condition [10]. Intuitively speaking, each *direct causal dependency* between variables of the model implies a corresponding directed arc in the BN.

Figure 2 depicts a simplified Bayesian network model of success. The variables of this simplified model—effort, ideas, success, and happiness—are all binary: either you are happy or not. The probability of success depends on both, effort and ideas. Effort indirectly affects happiness via success, but there is no *direct* causal influence. Therefore, if you are definitely a successful person, making a change in effort won't affect your 60% probability of happiness: Happiness is conditionally independent from effort, given success. If there was a direct connection between effort and happiness, we would have to add an extra arc between them.

Note how the conditional probability tables specify, for each node, the probability of each outcome, for all combinations of outcomes of their parent nodes. For effort and ideas, which have no parents in our model, we need to specify the unconditional a priori probabilities. The uniform 50-50 distributions are appropriate to express complete ignorance about the unconditional likelihood of effort or ideas.

3.2. Using Bayesian Networks for Simulation

The salient feature of a Bayesian network is its direct applicability for simulations. The network constitutes a complete probabilistic model of the variables in its domain and can, amongst other things, be used to solve "what-if?" scenarios. A simple simulation on a BN consists of the following steps:

1. The user provides *hypothetical evidence* $Y_1 = c_1, \ldots, Y_n = c_n$ for some selected nodes of the network[3], i.e., he fixes the value of some variables to one of their possible outcomes.

[3]Note that this hypothetical evidence is provided as part of a query. The user is not required to provide such information during the model construction. Also, it is possible to have queries without any hypothetical evidence (where $n = 0$).

2. A *belief update* is performed on the network[4]. During this operation, the conditional probabilities $P(X = x|Y_1 = c_1, \ldots, Y_n = c_n)$ are evaluated for all network variables X. Note that $c_1, \ldots, c_n$ are constant values. The conditional probabilities $P(X = x|Y_1 = c_1, \ldots, Y_n = c_n)$ thus represent the probability distribution of random variable X under the *user-defined scenario* $Y_1 = c_1, \ldots, Y_n = c_n$).

3. The user reads the probability distributions of his interest from the model.

For illustration, consider, once again, the Bayesian network from Figure 2. Assume that the user wants to perform the *diagnostic query* "How likely is it that a happy person is putting in some effort?" By setting the evidence of the happiness node to true and running a belief update over the network, the user can obtain the numeric answer, which is "55%", i.e., slightly higher than the corresponding a priori likelihood of "50%". A detailed explanation of the corresponding calculations is out of the scope of this work, but for an intuive explanation, consider that fixing happiness to "true" leads to an increase of the likelihood of success, via the direct link between those nodes. The increased likelihood of success, in turn, yields an increase in the likelihood of both, effort and ideas.

We are convinced that such a mechanism for performing simulations of user-defined "what-if?" scenarios is a highly desirable and useful tool in the hands of an engineer who is performing a timing analysis of a given system.

3.3. Probabilistic Interpretation of Abstraction

A segment (E, X) summarizes the timing behavior of all SMM paths from E to X by collecting the execution times of these paths in a multi-set. Interpreting relative frequencies as probabilities, we obtain, for each segment, a random variable on the possible execution times.

The random variables for different segments are generally not conditionally independent. Rather, they show some degree of correlation; a result of the fact that the concrete SMM path taken through a segment during an actual run of the software depends on the concrete SMM state through which the segment is entered, and thus on the concrete SMM path that was taken through previous segments.

At our level of abstraction, the correlations between the execution times of basic blocks can be viewed is being established via a "hidden" information channel. For example, an instruction cache can act as a mediator between the timing variables of two basic blocks that share a common cache line, creating a probabilistic dependency between them.

Once we have identified such information channels, we can create a Bayesian network model that incorporates the corresponding probabilistic dependencies.

3.4. Deriving the Network Structure

Besides performing the segmentation of the ASMM, we also have to identify the relevant dependencies between segments.

Let G be a segment. The *context set* $cs(G)$ of G is the set of all segments on which G *causally* and *directly* depends.

[4]Various different algorithms that perform this task have been proposed and implemented [9, 6, 3, 14].

For our application, the restriction to *causal* dependencies means that $cs(G)$ can only contain ancestor segments (w.r.t. the flow of control) of G. The restriction to *direct* dependencies means that transitive dependencies are not modeled explicitly.

Technically, the variables associated with the context set of $cs(G)$ will later form the Markov boundary of the variable associated with G.

The identification of context sets should generally be done in parallel to identifying segments, as both choices depend on each other.

Given a segment G, how can we determine the context set $cs(G)$ of G?

It is important to note that we cannot provide a feasible method for automatically identifying all dependencies of G. Rather, we propose a best-effort approach that yields a good approximation of $cs(G)$ by considering *candidate segments* that are likely to exert a strong influence on G.

Our primary source for candidate segments is our abstract knowledge about the hardware architecture and software semantics.

For example, if we have some knowledge about the instruction cache and memory layout of our target architecture, then we might be able to identify segments that are in conflict through the sharing of a common cache line.

Another example is the consideration of pipelining effects over segment borders: in a pipelined architecture, a segments potentially depends on its immediate predecessors through the shared pipeline state at the common segment boundary.

Our third example is the analysis of control flow dependencies. In the following we give a sketch of how we can identify candidate segments for $cs(G)$ through the use of use-definition chains.

Let x be a program variable that is used inside the condition of a control flow statement within segment G. Since the value of x can influence the flow of control in G, it also has a potential influence on the execution time of G. Next, consider the corresponding assignments of x (which can be easily obtained by static program analysis). If such an assignment of x is contained in at least one branch of a control flow statement St of segment G', then G depends on G' by way of the "hidden" common dependencies of both segments on the condition of St.

3.5. Classifying Execution Times

In a Bayesian network, we need to specify, for each variable X, the conditional probabilities w.r.t. all parent variables. If we have n possible outcomes (here: different execution times) for each segment, then the CPT for a segment with a context set size of m requires n^{m+1} entries. It is thus clear that we have to limit both, the size of the context set for each segment, and the number of outcomes for each variable.

The size of the context set of a segment can be reduced by considering only the strongest dependencies. The strength of a dependency can, for example, be judged by the *measure of mutual information* [13] of corresponding variables.

On the other hand, the number of outcomes for a variable can be reduced by classifying values. For example, if the possible execution times for segment G are in the range of 100 to 750 microseconds, then we might summarize the possible outcomes as intervals $[100, 250)$, $[250, 500)$, $[500, 750]$, plus a special outcome "null" that represents the situation where a segment is not executed. A reasonable classification is crucial for obtaining a significant probabilistic model.

3.6. Parameterizing the Network with Measurements

Once we have identified segments and their associated context sets, we have to parameterize the network with appropriate conditional probabilities of execution times. As mentioned in Section 3.3, our probabilities are based on relative frequencies.

We obtain the relative frequencies of execution times by performing measurements on the real computer system. The main advantage of our measurement-based approach is that we obtain our timing data from the actual system, which incorporates all the numerous and possibly peculiar implementation details and quirks of the future production system.

To parameterize our network, we need to obtain, for each segment G, the *conditional relative frequencies* $f(T = c \,|\, T_1 = c_1, \ldots, T_n = c_n)$ of execution times, where $T, T_1, \ldots, T_n$ are random variables representing the execution times for the segment G, and its corresponding context segments $\{G_1, \ldots G_n\} \in cs(G)$, and where $c, c_1, \ldots, c_n$ are the corresponding classes of execution times. This can be achieved as follows:

1. Generate test data vectors $d_1, \ldots, d_m$, that force a flow of control through G. This can be done by harnessing techniques like random test data generation and model checking [11].
2. For each test data vector d_i, $1 \leq i \leq m$, measure the execution times $t_{i,G}, t_{i,G_1}, \ldots, t_{i,G_n}$ of $G, G_1, \ldots, G_n$. Available options for this step are the use of intrusive techniques like static source code instrumentation, or non-intrusive techniques like timing trace generation with hardware trace probes.
3. Obtain the *joint absolute frequencies* as

$$F(c, c_1, \ldots, c_n) = |\, \{i \,|\, t_{i,G} \in c,\ t_{i,G_1} \in c_1,\ \ldots,\ t_{i,G_n} \in c_n,\ 1 \leq i \leq m\} \,|,$$

   and subsequently the conditional relative frequencies as

$$f(c \,|\, c_1, \ldots, c_n) = \frac{F(c, c_1, \ldots, c_n)}{\sum_{x \in domain(T)} F(x, c_1, \ldots, c_n)}.$$

To obtain representative frequency distributions for the segment, the test data vectors $d_1, \ldots, d_m$ should throughly cover the timing behavior of G. Since a full coverage of all possible SMM paths is usually infeasible, we confine ourselves to up to k random samples per ASMM path, i.e., we try to achieve full path coverage of the abstract model within the segment and try to generate up to k unbiased measurements per path. Full ASMM path coverage can be achieved by applying techniques like random test data generation and model checking [11].

T_{S_1}		10ms			11ms			null		
T_{S_2}		20ms	21ms	null	20ms	21ms	null	20ms	21ms	null
T_S	30ms	1	0	0	0	0	0	0	0	0
	31ms	0	1	0	1	0	0	0	0	0
	32ms	0	0	0	0	1	0	0	0	0
	null	0	0	0	0	0	0	0	0	1
	inconsistent	0	0	1	0	0	1	1	1	0

Table 1. Example CPT that relates the execution time of two sequential segments S_1 and S_2 with the execution time of their corresponding super-segment S

3.7. Multiple Layers of Abstraction

The modeling we have described so far produces Bayesian networks that capture the running times of individual segments. However, the engineer will usually also be interested in the execution times of larger program sections, in particular the execution times of the whole program (most specifically in the overall WCET). To this end, we introduce multiple layers of abstraction to our model.

Above the basic segmentation layer, we introduce a coarser *super-segmentation* layer, such that the basic segmentation layer can be seen as a refinement of the coarse layer. The network corresponding to the super-segmentation layer is, however, not parameterized by measurements, but the values of these nodes depend functionally on their corresponding basic segments.

Such functional dependencies can by modeled as "deterministic" conditional probability tables, i.e., CPTs that contain only zeros and ones. Table 1 shows an example CPT that relates the execution time of two sequential segments G_1 and G_2 with the execution time of their corresponding super-segment G. Note the outcome "inconsistent", which is an artifact of our modeling approach. We must include such hypothetical situations in our model to capture anomalous flows of control between segments that violate structural flow constraints [8]. However, additional consistency nodes can easily reduce the actual probability of these outcomes to zero.

The final BN model includes several layers of segment abstraction, where the top layer contains only a single segment that represents the overall behavior of the system. The corresponding variable captures the execution time of the complete system, including the empirical, probabilistic worst case outcome.

4. Related Work

Lemeire and Dirkx [7] present an approach for performance analysis of concurrent systems that is based on Bayesian networks. Whereas the structure of our networks is based on the structural properties of the system under test, the approach of Lemeire and Dirkx is based on a functional description of the system that requires system-specific knowledge. In contrast to this, our approach is generic. Eventually it should be possible to perform all steps, i.e., construction of the ASMM, segmentation, identification of candidate dependencies, and parameterization, automatically.

Bernat et al. [1, 2] present a probabilistic approach for WCET calculation where the execution time frequency distributions of different code sections are combined by one out of three combination operators. The choice of the operator depends on whether the distributions are correlated via a known joint distribution, correlated via an unknown joint distribution, or uncorrelated. The operators are used to calculate an overall frequency distribution for the whole program. Compared to this operational approach, which is targeted at the derivation of one particular *static distribution*, our approach

is aimed at the derivation of an *interactive timing model* of the system under test, on which the user can perform arbitrary simulations. Obtaining the overall execution time distribution over all possible inputs (and thus obtaining a probabilistic estimate of the WCET) is only one particular use case of such a timing model.

The concept of ASMM segments that we introduce in this paper is a generalization of the concept of CFG segments introduced in [11, 12].

5. Future Work

In Section 2.4 we introduced the concept of segments and indicated their application. We have yet to provide a concrete segmentation algorithm.

In Section 3.5 we argued that we have to limit the number of possible outcomes of random variables, and proposed classification as a solution. We are currently working on a WCET-aware classification method for execution times that tries to minimize the loss of information that is relevant for WCET calculation.

In Section 3.7 we have presented a brief description of how we can include multiple layers of model abstraction into a single Bayesian network model. Further work is needed on this concept.

The Bayesian network structure that we propose in this paper models variables for segment execution times and their dependencies. We are currently developing a scheme for deriving much richer network models that expose conditions on program variables and the flow of control. These models will feature clearer and hopefully more intuitive dependency structures and richer choices for simulation.

We are planning to integrate the presented approach in the timing analysis suite that is currently being developed within the FORTAS project[5]. Within the project, we are developing a fine-grained abstract system machine model, where states will carry more information than merely a basic block number. Augmenting and adapting the presented concepts to this model will be a challenge for future work.

Also, in acknowledgment that our ideas need to be tested within quantitative experiments, we are planning to provide a working implementation of our approach within the FORTAS framework.

6. Summary and Conclusion

In this work, we have presented a probabilistic approach for modeling the execution time of software-based embedded systems that is based on the framework of Bayesian networks. The structure of our networks, which represents the conditional dependencies between execution times, is derived from knowledge about the hardware architecture and software semantics, where is the parameterization is obtain by performing measurements on the real physical system.

The salient benefit of having a Bayesian network model of the timing behavior of the system under test is its ability to let the engineer perform simulations, like, e.g., "what-if" timing scenarios. We are convinced that this ability provides a highly desirable and useful tool to the hands of an engineer who

[5]The FORTAS project is a cooperation between the Real Time Systems Group at the TU Wien and the Formal Methods in Systems Engineering Group at the TU Darmstadt, with the goal of developing a software engineering oriented timing analysis method that integrates measurement-based and formal methods. Please c.f. `http://fortastic.net/`.

is performing a timing analysis of a given system.

In particular, the querying abilities of the BN model subsume unconditional queries for the total execution time of the underlying system. Such queries return a probability distribution of total execution times. The upper bound of that distribution is a probabilistic estimate of the WCET of the system. Thus, our approach is more general, and provides a broader setting for timing analysis then pure WCET calculation.

7. Acknowledgment

We would like to thank the reviewers for providing many useful hints on how to improve the present paper and pointing out the aspects of our approach that require particular attention in our future work.

References

[1] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[2] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pwcet: a tool for probabilistic worst case execution time analysis of real–time systems, 2003.

[3] Jian Cheng and Marek J. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[4] Raimund Kirner and Peter Puschner. Obstacles in worst-cases execution time analysis. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Orlando, Florida, May 2008.

[5] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer, 1997. ISBN: 0-7923-9894-7.

[6] Steffen L. Lauritzen and David J. Spiegelhalter. *Readings in uncertain reasoning*, chapter Local computations with probabilities on graphical structures and their application to expert systems, pages 415–448. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[7] Jan Lemeire and Erik Dirkx. Causal models for parallel performance analysis. In *Proc. 5th PA3CT Symposium*, Edegem, Belgium, Sept. 2004.

[8] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[9] Judea Pearl. Fusion, propagation, and structuring in belief networks. *Journal of Artificial Intelligence*, 29(3):241–288, 1986.

[10] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann Publishers, San Mateo, CA, 1988. ISBN: 0-934613-73-7.

[11] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*, Munich, Germany, Mar. 2005.

[12] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.

[13] Yiyu Yao. *Entropy Measures, Maximum Entropy Principle and Emerging Applications*, chapter Information-theoretic measures for knowledge discovery and data mining, pages 115–136. Springer, 2003.

[14] Changhe Yuan and Marek J. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *Proc. 19th Annual Conference on Uncertainty in Artificial Intelligence*, 2003.